Script-Free HTML: Preventing Cross-Site Scripting While Permitting HTML-Rich Content

Undergraduate Research Thesis

Presented in Partial Fulfillment of the Requirements for the Degree B.S. Computer Science and Engineering with Honors Research Distinction in the College of Engineering at The Ohio State University

By: Matthew Seffernick

B.S. Computer Science and Engineering

The Ohio State University

2013

Committee:

Paul Sivilotti, Advisor

Michael Bond

I. Abstract

When someone visits a web site, the site's server uses input from the person's web browser to dynamically generate the webpage returned to the user. If hackers can find a weakness in the site's code and control how webpages are generated, they can insert their own scripts into the webpage returned to visitors. These scripts run in the visitor's browser and can compromise the visitor's personal information. The injection of scripts into a webpage by means of evading input filtering is called a cross-site scripting (XSS) attack. Even popular websites, such as Google, Facebook, and YouTube, have been exploited by XSS attacks (KF & DP, 2012). In 2010, XSS attacks were ranked the 2nd-leading source of web security risk (OWASP, 2010).

XSS attacks, by their very nature, are not detectable client-side (e.g., by web browsers or antivirus programs). Current methods to prevent XSS exploits are either ineffective (allowing some attacks to succeed) or overly prohibitive (preventing legitimate HTML-rich content). This project describes a new approach: The structure of safe input is rigorously defined and a server-side tool is implemented to detect the presence of a potential XSS attack. This tool prevents XSS attacks while still permitting HTML-rich content. We define a new context-free grammar (Script-Free HTML 4) that precisely characterizes safe

input. Our approach is evaluated by applying it to a benchmark of known XSS vulnerabilities. We also consider the future evolution of this approach in the ever-changing world of web standards.

II. Acknowledgments

I would like to thank Paul Sivilotti, my research advisor, for all the time and experience he's contributed to making my research successful.

III. Vita

2009.....Honda-OSU Math Medal Award

July 2010.....Celina High School

2010..... AP Scholar with Distinction

2010-Present .....Ohio State University Honors

2010-Present .....Maximus Scholarship

2010, 2011..... Thomas L. Thomas Scholarship

June 2012-August 2012 ..Cisco Choice University Engineer Intern, Cisco Systems

2013..... Lumley Engineering Fund

Fields of Study

Major Field: Computer Science and Engineering

IV. Table of Contents

V. Table of Figures

1.0 Introduction

1.1 Background and Motivation

It is no secret that our lives are intertwined with the Internet. Among many other things, we use the internet to look up directions, talk with friends and family, buy books, do our banking, and collaborate with co-workers. The internet has not only increased convenience in our lives, but has also made possible the otherwise impossible. Just like in person, we have to communicate information to make these things happen. Examples of such information include our addresses, the identity of our friends, the conversations we have, our credit card numbers, our usernames and passwords, and our confidential work files. To take advantage of the convenience the internet offers, however, we must transfer our sensitive information over the internet.

Although legitimate websites attempt to safeguard our information, black-hat hackers actively seek to bypass those safeguards and steal our information for personal gain. For instance, one might steal your name, address, credit card number, and social security number to steal your identity and money. Black-hat hackers use a variety of tools and tricks to help them steal your sensitive information. For example, some hackers will find your username, which tends to

be publically available, and attempt to guess every password you could possibly have. This method, called a brute force attack, may sound slow and tedious, but hackers have computer programs do all the guessing for them, and can guess thousands of passwords per second. Some hackers will make a website that imitates another website. They will then try to trick you into logging into their website as if it were the original, giving them your username and password. This particular example is called a phishing scheme, while the setup that lures you to the website is called social engineering (OWASP, 2012). Some attackers take advantage of flaws in code itself. Although finding flaws in code, or vulnerabilities, may be more difficult to do, such an attack does not require any of your information or for you to do anything out of regular behavior. In September 2012, a zero-day exploit in recent versions of Internet Explorer gave hackers full access to peoples' computers (Oremus, 2012). Similarly, some hackers exploit website code. All it took to become infected with the Sammy MySpace worm was to visit an infected webpage (Kotadia, 2005).

When you visit a website, your browser requests a webpage from the remote web server. The server takes this request, does some computing work, and responds by sending a file containing code detailing what to display on the page and what to do if you interact with the page. The code detailing how to make the page is called a markup language, commonly HyperText Markup Language (HTML), and the code detailing how to react to your actions is called a script, commonly

JavaScript. Sometimes, the HTML you get back will produce a page displaying information you supplied. For example, if you run a Google search that yields no results, Google will print your search terms back to you. To display that text, the server copies your input into your response HTML.

Hackers exploit response HTML by giving websites input that, when read by a web browser, executes malicious scripts. This form of attack is called an XSS attack, and the malicious input they create is called an attack vector (Shar & Tan, 2012). The primary obstacle for an attack vector is input-checking code on the web server. If hackers can find a weakness in the server code so their input will execute on your web browser, they can manipulate your web browser into doing their bidding. For instance, you can fall victim to an XSS attack simply by following a URL containing an attack vector. Forums and wikis consist almost entirely of user input. Thus, you can also fall victim to an XSS attack simply by loading a web page containing an attack vector.

In 2010, OWASP ranked XSS attacks 2nd in their top 10 "Most Critical Web Application Security Risks" (OWASP, 2010). In 2011, Mitre ranked XSS attacks 4th in their top 25 "Most Dangerous Software Errors" (Mitre, 2011). Because this attack method can affect anyone that browses the internet, the solution to this problem benefits, among many other people, anyone who uses online banking sites, social networking sites, and webmail.

1.2 Contribution

To address the problem of XSS attacks, we develop: a context-free grammar, called SFH4, which produces a language that follows the structure of HTML and is free from JavaScript invocations; a parser for SFH4; and a methodology for generating a context-free grammar and parser from a Document-Type Definition (DTD).

SFH4 determines if input is safe or unsafe on the basis of the input's structure and scripting content. For instance, if a user inputs well-formed HTML 4 with no signs of possible script invocations, the parser accepts it as safe. Conversely, if a user inputs malformed HTML 4 or content which follows a pattern known to risk browser script invocation, the parser rejects it as potentially unsafe. Thus, the parser is sound but imprecise; it only accepts safe input, but also rejects input which may, in practice, be safe. Web applications can use an SFH4 parser to test the safety of raw or filtered user input before embedding it in returned HTML.

1.3 Thesis Outline

Chapter 2.1 defines XSS attacks and discusses how they work. Chapter 2.2 presents existing methods for preventing XSS attacks, while chapter 2.3 discusses each method's strengths and weaknesses to explain where these methods excel and where issues remain.

Chapter 3.1 contains an in-depth summary of the project's solution, which is to use a Script-Free HTML 4 (SFH4) parser on the web server to detect unsafe user input, while chapter 3.2 discusses the process used to implement the SFH4 parser. Chapter 4.1 discusses the findings of designing and implementing SFH4 and its parser, Chapter 4.2 presents results from testing, and chapter 4.3 discusses future work.

2.0 Literature Review

2.1 Problem Analysis

To research the overview of and current methods of handling XSS attacks, we reference an article in the magazine **COMPUTER** (Shar & Tan, 2012), papers on code injection and XSS (Bisht & Venkatakrishnan, 2008) (Ligatti & Ray, 2012), and material published by The Open Web Application Security Project (OWASP, 2012).

Ligatti and Ray define code-injection attacks as attacks in which an attacker's input is used as code (Ligatti & Ray, 2012). Shar and Tan define XSS attacks as attacks in which web browsers treat a malicious user's input, also known as an attack vector, as scripting content (Shar & Tan, 2012). As these definitions demonstrate, the identification and prevention of XSS attacks requires two things: an understanding of when user input may be interpreted as browser scripts and how attackers bypass preventative measures to inject scripting content into pages served to others.

Based on the HTML 4.01 Strict DTD, browsers should only invoke scripts in script elements and event attributes (Hors, Jacobs, & Raggett, 1999). For an example of these script invocation methods, see Figure 1.

```
<script>alert('script tag');</script>
<a href="about:blank" onhover="alert('event
attribute');">Hover over me!</a>
```

Figure 1: Script Invocations

As highlighted by Bisht and Venkatakrishnan, however, browser quirks introduce major difficulty and uncertainty in identifying potential scripting content (Bisht & Venkatakrishnan, 2008). For example, some versions of Internet Explorer invoke JavaScript when "javascript:" starts the IMG element's src attribute. For an example of such an element, see Figure 2.

```
<img src="javascript:alert('browser quirk')">
```

Figure 2: Browser Quirk

Additionally, embedded content, such as Cascading Style Sheets (CSS) and Flash files, can contain browser-side scripting content and is impossible to detect without also scanning the embedded content (OWASP, 2013). Thus, to prevent scripting invocations based on only a web page's source HTML, it becomes necessary to prevent input from becoming involved with script elements, event attributes, and content embedding.

As demonstrated by OWASP's filter evasion cheat sheet examples (made by RSnake), the key to an attack vector's success is its ability to open and close elements and its ability to start and end attributes. To introduce scripting content, an attack vector commonly must close the element or end the attribute in which it is originally used and open a new element or start a new attribute (OWASP,

2013). With no restraints, this task is trivial for an attack vector; to end an attribute, an attack vector needs only to include a quotation mark, and to close an element, an attack vector needs only to include the element's closing tag. As a result, unchecked input leaves a site completely vulnerable to XSS attacks. For an example of an attack vector which demonstrates an attempt to end attributes and close tags, see Figure 3.

```
';alert(String.fromCharCode(88,83,83))//';alert(String
.fromCharCode(88,83,83))//";
alert(String.fromCharCode(88,83,83))//";alert(String.f
romCharCode(88,83,83))//--
></SCRIPT>">'><SCRIPT>alert(String.fromCharCode(88,83,
83))</SCRIPT>
```

Figure 3: Attack Vector

However, the more restrained the input, the fewer ways the input may be used legitimately. For instance, successfully forcing each character of user input to be printed by the web browser prevents user input from being used to format text or create page structure. Such features may be useful on wikis, blogs, and forum posts.

2.2 Prevention Methods

Shar and Tan highlight 3 simple, yet popular and effective, methods for preventing XSS attacks: blacklisting, whitelisting, and escaping characters. Blacklists describe unsafe input that should be rejected when encountered. Typically, web applications blacklist attack vectors by describing them with

regular expressions and scanning input strings for matches. When the application encounters a match, the application typically either removes the matched substring or rejects the entire input string. Whitelists, on the other hand, define safe input; web applications reject input that does not match values in its whitelist. Escaping characters replaces all HTML meta-characters in user input with their HTML-encoded equivalents. That is, any character in user input which would normally have special meaning is replaced with text which instructs web browsers to display the character rather than interpret it as page structure (Shar & Tan, 2012).

XSS-Guard generates a webpage twice, once with user input and once with safe input following the same path through code. Using a parser based on FireFox's content sink, XSS-Guard generates a JavaScript parse tree for each page and compares the parse trees. If XSS-Guard finds that the parse trees are syntactically equivalent, XSS-Guard considers the generated output safe and does nothing; otherwise it alters the scripting content sent to the user, replacing what it identifies as malicious scripting content with a benign counterpart. Thus, XSS-Guard uses the actual scripting content generated to determine safeness rather than making decisions based on user input or the output page alone (Bisht & Venkatakrishnan, 2008).

HTML Purifier takes an approach similar to that of SFH4; it uses the structure of HTML and a whitelist of HTML elements and attributes to maintain a description of safe input. The application parses input into tokens, alters and validates input according to its settings, and converts the resulting tokens back into a string for use by the web application. HTML Purifier intends for web applications to take user input, pass the user input through HTML Purifier, and embed the returned text straight into the output page (Yang, 2012).

2.3 Analysis of Methods

Due to their simplicity, blacklists, whitelists, and character escaping are all efficient to use and easy to apply. However, as Shar and Tan indicated, they are flawed. Blacklists tend to fail to catch all attack vectors, whitelists prohibit much valid input, and character escaping prevents use of HTML-rich input (Shar & Tan, 2012). In addition, "successful" use of these methods typically means exclusively printing user input, not using it for any form of page structure. As a result, web developers may not be able to use these methods to permit users to complexly format their own content or build their own pages.

XSS-Guard more effectively detects scripting content due to its high-level approach, but at the cost of generating and comparing two JavaScript parse trees and correcting output to make it safe. Additionally, XSS-Guard fails when web applications use conditional copying, such as when one string is copied character

by character to another string. XSS-Guard also fails when an exploit is "embedded in a Flash object included by the web application" (Bisht & Venkatakrishnan, 2008).

HTML Purifier shares many of its strengths with SFH4 due to their similar approaches. For instance, enforcing a strict HTML structure and an element and attribute whitelist effectively stops most risky user input. However, HTML Purifier's behavior is difficult to reason about due to its editable whitelist and lack of clear detection and editing definitions. Additionally, HTML Purifier doesn't sanitize input based on the context in which the input will be used, potentially leading to unsafe use of "safe" input, an issue discussed by Ligatti and Ray.

2.4 Related Work

SQL injection attacks are a web application security problem similar to XSS attacks. In an SQL injection attack, an attacker sends an input string to a web application with the intent of the web application executing the string as code in an SQL statement rather than as a literal value (Su & Wassermann, 2006). As a result of SQL injection attacks, attackers may steal information from the website (such as obtaining account details and login credentials) or remove information from the site (such as by dropping a database).

An approach to detect and prevent these attacks at run-time is to create and compare SQL query parse trees (Buehrer, Weide, & Sivilotti, 2005). A parse tree represents the structure of an SQL statement, where leaves of the tree represent specific tokens (keywords, identifiers, and literals) and nodes of the tree represent groups of tokens. Assuming web applications always intend for user input to be a proper subtree in an SQL parse tree, the parse tree generated as the result of an SQL injection attack necessarily will not match that of the parse tree generated by the SQL query intended by the web application.

To apply a similar approach to the prevention of XSS attacks, we would compare JavaScript parse trees rather than SQL parse trees. However, webpages sent to visitors consists of HTML, which can invoke JavaScript. Thus, we can't work with pure JavaScript and must work with HTML parse trees instead. Then our task becomes determining what subtrees of any HTML parse tree invoke JavaScript, and when the HTML parse tree of user input contains a subtree that invokes JavaScript. Thus, we can see that this approach helps us gain insight on how to detect XSS attacks, but cannot be applied directly to prevent it. If we color the parts of an HTML parse tree which invoke JavaScript, our task is to restrict user input to uncolored proper subtrees of an HTML parse tree.

An approach to detect SQL injection vulnerabilities before run-time is to first describe all the strings used in SQL queries that may be influenced by user input

run through server code, then determining if the resulting grammar can generate an SQL injection attack (Wassermann & Su, 2007). In their implementation, Wassermann and Su use context-free grammar rules to describe the changes a string may undergo by PHP operations and methods. Once the grammar for the given code is generated, they test if the grammar can produce a syntactically open statement, such as if a query string can contain an odd number of un-escaped quotation marks.

The overall goal of this approach is to prevent attacks by detecting when user input can result in syntactically open strings. While we could use such an approach to prevent user input from manipulating HTML elements and attributes, and thus prevent many XSS attacks, it would prevent user input from being used to style text and structure pages. We do see, however, that use of context-free grammars can help us reason about what a page with user-influenced input may look like, and thus help us reason about the presence of vulnerabilities.

3.0 Discussion

3.1 Solution Overview

To bypass filters and invoke browser-side scripts, most successful attack vectors use a combination of: malformed HTML, particular HTML elements and attributes, and browser quirks. We prevent XSS attacks by restricting user input to well-formed HTML consisting of white-listed HTML elements and attributes. We captures this description by defining a context-free grammar, called SFH4, and implementing a parser for it through the use of the HTML 4.01 Strict DTD (Hors, Jacobs, & Raggett, 1999) and GNU's Flex (The Flex Project, 2008) and Bison (Free Software Foundation, Inc., 2008) programs. The resulting parser accepts ASCII text as input and returns true if and only if the input is a member of SFH4.

To use the SFH4 parser, a web application must do two things. Firstly, the application must construct a test web page using the user's input and mimicking its use in the actual webpage. For instance, if an application intends to embed user input inside an HTML "div" element, the application must first create a test page which starts with the content in Figure 4, followed by the user content, and ending with the content in Figure 5. The only important surrounding information

for generating the test page is the HTML structure surrounding the user input; other elements and character data are not important.

```
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Transitional//EN">
<html><body><div>
```

Figure 4: Test Page Prefix

```
</div></body></html>
```

Figure 5: Test Page Suffix

Secondly, the application must pass the test page to the parser. If the parser rejects the test page, then the web application should handle the user input as if it is unsafe to use as the application intended. Otherwise, the application may use the user input as it intended. The parser may reject input that is in fact safe, but further processing would be necessary to determine whether or not the input is, in fact, safe. Whether or not such input really is safe or not may vary from browser to browser, as well, due to browsers responding differently to the same web pages, particularly those with non-standard structure.

A server using the parser executable will need to locate and set permissions on the executable such that the parser is reachable and executable by the web application. The web application can then call the executable, redirecting stdin to a file containing the test page to be parsed. If the program exits with status 0, then the parser considers the test page safe. Otherwise, the parser considers the

test page unsafe. Because the parser uses a file for input, multi-threaded use of the parser requires use of multiple files as potential input buffers.

3.2 Implementation Process

For our implementation, we used the HTML 4.01 Strict DTD as input, mIRC Scripting Language (MSL) from mIRC v7.17 for generating token and grammar definitions, GNU Bison v2.4.1 and GNU Flex v2.5.4 to generate the C code for the parser, and Gnu C Compiler (gcc) v4.6.1 to compile the parser code. To summarize the process before going into detail, we take a Document Type Definition (DTD) as input, and in steps 1 and 2 parse the DTD to create a whitelist of permitted elements and attributes. In step 3, we alter the whitelist of elements and attributes to exclude the elements and attributes we deem unsafe. In step 4, we generate the grammar using the whitelist of elements and attributes and a pre-written definition of tokens. In the remaining steps, the Bison and Flex applications generate C code for the parser using our grammar definition, and gcc compiles the C code into the final executable.

Firstly, an MSL script (`parseItems`) parses the DTD for element, attribute, and entity definitions. Secondly, more MSL scripts (`evaluateEntities`, `evaluateElements`, `evaluateAttributes`) evaluate the definitions and parse them into two lists: a list of elements and their permitted sub elements, as well as a list of elements and their attribute declarations.

In the third step, another MSL script (`removeItems`) modifies the two lists to remove unsafe elements and attributes and restrict the values of attributes with known browser quirks. Rather than removing elements and attributes from the parsed data, we could have removed element and attribute definitions from the DTD before passing the DTD to the parsing process. For a diagram of steps 1-3, see Figure 6.

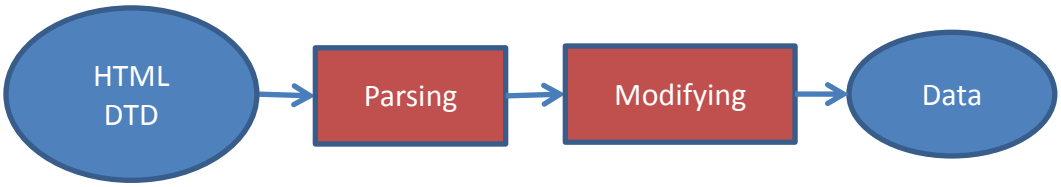

Figure 6: Parsing and Modifying

With exceptions from browser quirks, attack vectors invoke JavaScript with unsafe attributes, not particular values within otherwise safe attributes. As such, we default the value type of all attributes to "PCDATA," and make special exceptions for known browser quirks. By defaulting value types to PCDATA, we are able to make the language less complex and reduce the manual intervention necessary for the process workflow to complete.

In the fourth step, another MSL script (`makeInputFiles`) reads the parsed data and manually created configuration files to generate the token and grammar definitions. For a diagram of this step, see Figure 7. For the complete set of MSL scripts, see Appendix E.

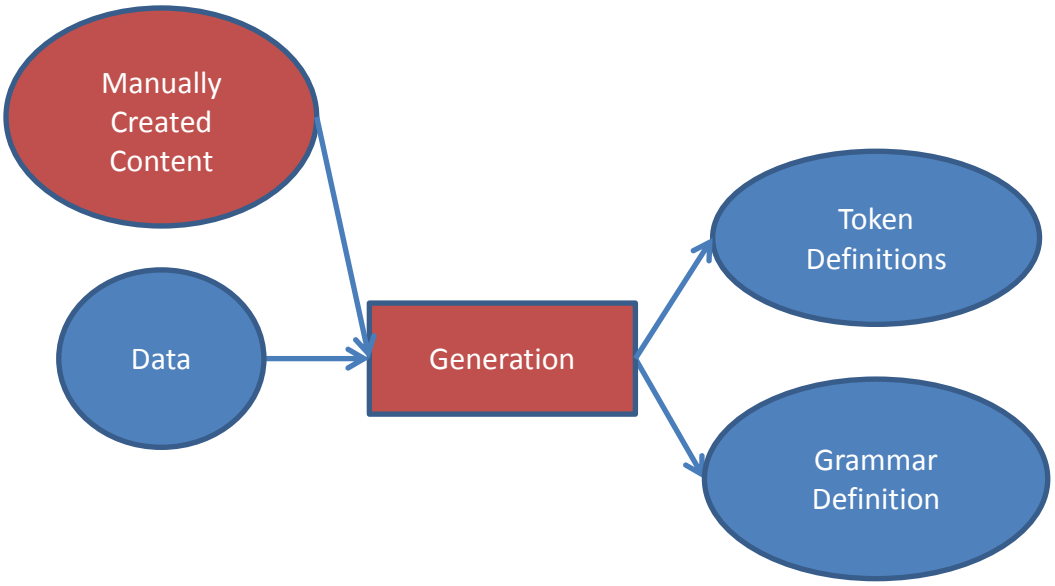

Figure 7: Definition Generation

To generate the token definitions, the MSL script reads from a configuration file including pattern definitions, match-handling, and procedures written in C. For the scanner configuration file, see Appendix A. Basic pattern definitions, such as SYMBOL, NUM, and ALPHA, are used to make more complex patterns, such as PCDATA and URI. The match-handling rules consist of patterns and C code and dictate how the Flex-generated scanner should respond to characters from its input stream to determine a token type to return. In our parser, the rules instruct the scanner to ignore white space and to always treat the greater than and less than symbols as tokens.

Global variables `_inOpenTag` and `_inQuote` are checked and updated with each pattern match to determine if the matched content is inside a tag or attribute

value before deciding what token type to return. The procedures in the configuration file include `printDebug`, `upperString`, and `parseID`. `printDebug` outputs the text received and its token type, and `upperString` converts input text to uppercase. MSL procedurally generates the function `parseID`, which checks ID values for white-listed element and attribute names. For an abbreviated copy of the Flex input generated in this step, see Appendix B.

To generate the grammar definition, the MSL script reads two configuration files containing token definitions, method prototypes, a start rewrite rule, and C methods required by Bison. For the grammar configuration files, see Appendix A. Manually defined tokens include PCDATA, URI, ERROR, and DOCTYPE. Because our pipeline does not make any guarantees regarding the order in which it produces SFH4 rewrite rules, the configuration files contains a start rule to ensure a correct starting rule.

Bison requires three C methods for it to function: `yylex`, `yyerror`, and `main`. `yylex` is implemented by Flex, so the configuration file only includes a prototype of the method. `main` and `yyerror`, on the other hand, are completely defined in the configuration files.

The MSL script procedurally generates ID token declarations and all SFH4 rewrite rules. For each HTML element, MSL generates a starting rewrite rule, a sub-element rewrite rule, and an attributes rewrite rule. The starting rule includes

the start tag of the element, attributes inside its open tag, sub-elements after its open tag, and a close tag after its sub-elements. In the case where an element has no sub elements, the sub-elements and close tag are omitted. The sub-elements rewrite rule rewrites to any number of any of the elements permitted between the element's open and close tags, including PCDATA. The attributes rewrite rule rewrites to any number of any of the attributes permitted inside the element. For an abbreviated copy of the grammar definition, see Appendix D.

Next, in the fifth step, the process passes the token and grammar definitions to Flex and Bison, generating C code. Finally, the process compiles the C code with gcc to create the parser executable. For diagrams of these final steps, see Figure 8 and Figure 9.

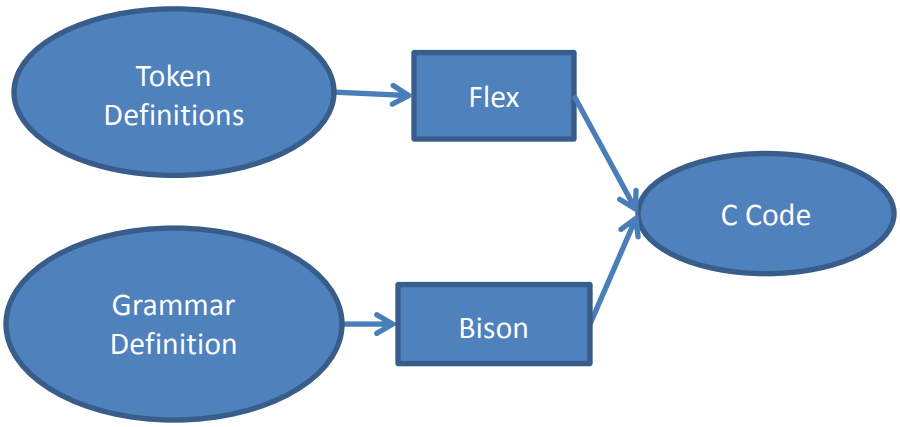

Figure 8: Code Generation

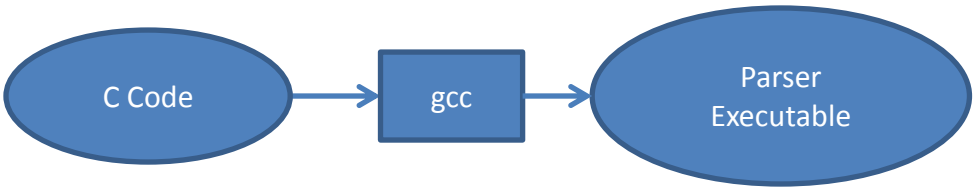

Figure 9: Code Compilation

4.0 Evaluation

4.1 Findings

Upon analyzing common attack vectors, we found that most attack vectors could be generalized to a few patterns. In general, attack vectors: violate HTML structure, embed foreign content, invoke scripts in CSS, or exploit browser quirks. Enforcing a strict HTML structure handles the case of poor input structure breaking surrounding HTML, and enforcing an element and attribute whitelist handles the cases of embedded foreign content and scripts invoked in CSS. Because browser quirks are prone to vary by version and brand of web browser, however, not only are browser quirks difficult to handle, but they are also impossible to predict. Thus, one important step to preventing XSS attacks is for web browsers to change to only invoke scripts in standardized cases.

As stated in Chapter 3.1, our approach expects web applications to construct test web pages mimicking the context in which the user input will be embedded. Thus, our approach is not particularly simple to use on its own and may be prone to human error from the web developer. A solution to this problem may be to package the parser executable with web development libraries that interface with the executable. For instance, web developers could import a PHP function for the

parser that does three things given the user input and an enumerated value indicating the context in which the user input will be used. Firstly, it automatically generates the prefix and suffix for the input. Secondly, it runs the input through the parser, encapsulating the call to exec. Finally, it returns a value for “safe” and a value for “unsafe” depending on the parser's exit status. Such a function should make the parser more convenient to use and the contexts in which user input has been validated easier to track. For an example of such a library, see Appendix F.

Additionally, our approach requires that user input be scanned for every context in which it's used. This is because text that is syntactically correct and safe in one context (such as between HTML "div" tags) may not be syntactically correct or safe in another context (such as inside the open tag of an HTML "div" element). Thus, validating input is not as simple as scanning it before storing it in a database or scanning input once and then using it in multiple locations. The only safe solution would be to validate input before each of its uses, or to track the contexts in which the input will be used and validate the input once for each context before use.

4.2 Test Results

Our approach was tested by implementing an SFH4 parser and running generic test cases against it, including cases that should pass and cases that should fail.

The 5 cases that should pass, based on the HTML 4.01 Strict DTD, tested the parser's acceptance of valid HTML, complex nesting, and variations in use of attributes. The 16 cases that should fail, based on OWASP's filter evasion cheat sheet, tested the parser's rejection of malformed HTML and known XSS attack vectors. Out of the 21 test cases, our implementation accepted 4 out of the 5 it should have accepted, and rejected all 16 it should have rejected. For the tests run and the results, see Table 1.

Table 1: Test Results

Test #	Expectation	Result	Description
1	ACCEPT	ACCEPT	Simple case (doctype, html; no attributes)
2	ACCEPT	ACCEPT	Nesting
3	ACCEPT	ACCEPT	Attributes
4	ACCEPT	REJECT	UTF-8 encoding, simple case
5	ACCEPT	ACCEPT	Permit PCDATA inside appropriate elements
6	REJECT	REJECT	No doctype (html, head, title)
7	REJECT	REJECT	Improper parent element (html, head, title, div)
8	REJECT	REJECT	Custom element (html, head, title, body, xss)
9	REJECT	REJECT	Attributes with no value
10	REJECT	REJECT	Stray cdata within tags (html, cdata, head, title)
11	REJECT	REJECT	Improperly closed attribute values
12	REJECT	REJECT	Script tags
13	REJECT	REJECT	Javascript in img src
14	REJECT	REJECT	On* attributes
15	REJECT	REJECT	Frame tags
16	REJECT	REJECT	Style element
17	REJECT	REJECT	style attribute
18	REJECT	REJECT	Meta-linked css file
19	REJECT	REJECT	Link-linked css file
20	REJECT	REJECT	Redirect 302
21	REJECT	REJECT	Custom attribute

In our implementation's failed test case, it rejected safe input which was encoded in UTF-8 rather than ASCII. This false positive was expected because our implementation did not take into account the various character encodings. To address this, the parser could have taken the character encoding as an argument to then read and compare text accordingly. In the case of character encodings that permit more characters than ASCII, the token definitions would need expanded to permit the extra characters.

The performance of our implementation was tested by running each of the verification tests and a large input test 20 times. The large input test contained 2.8KB of safe data inside valid HTML. The system used to perform testing had an 8-core Intel(R) Xeon(TM) CPU at 2.80GHz per core, and 8059128 kB of memory. The tests ran with a mean of 0.022 seconds per set of 22 tests with a variance of 0.004 seconds.

4.3 Future Work

The most obvious future work for this project is to extend SFH4 to HTML 5. This task could be accomplished simply by creating a DTD for HTML 5, modifying it to exclude unsafe elements and attributes, and then running the DTD through the process used in this project just the same as was done with the HTML 4.01 Strict DTD for SFH4. A similar approach might also be used to make a

Script-Free Cascading Style Sheets parser. We can also extend SFH4 to correct unsafe input rather than simply detect it.

5.0 References

Bisht, & Venkatakrishnan. (2008). XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks.

Buehrer, G., Weide, B., & Sivilotti, P. (2005). *Using Parse Tree Validation to Prevent SQL Injection.* The Ohio State University.

Free Software Foundation, Inc. (2008). *Bison - GNU Parser Generator*. Retrieved 2012, from GNU Operating System: http://www.gnu.org/software/bison/

Hors, A. L., Jacobs, I., & Raggett, D. (1999, December 24). *HTML 4 Document Type Definition.* Retrieved February 1, 2013, from W3C: http://www.w3.org/TR/html4/sgml/dtd.html

KF, & DP. (2012). *Cross Site Scripting (XSS) attacks information and archive*. Retrieved February 14, 2013, from XSSed: http://www.xssed.com/

Kotadia, M. (2005, October 17). *Samy opens new front in worm war.* Retrieved February 1, 2013, from CNET: http://news.cnet.com/Samy-opens-new-front-in-worm-war/2100-7349_3-5897099.html

Ligatti, & Ray. (2012). *Defining Code-Injection Attacks.* University of South Florida.

Mitre. (2011). *2011 CWE/SANS Top 25 Most Dangerous Software Errors.* Retrieved February 1, 2013, from Mitre: http://cwe.mitre.org/top25/

Oremus, W. (2012, September 18). *Hackers Just Found a Big Hole in Internet Explorer. Should You Switch Browsers?* Retrieved February 1, 2013, from Slate: http://www.slate.com/blogs/future_tense/2012/09/18/internet_explorer_zero_day_vulnerability_ie8_ie9_hack_means_you_should_switch_browsers_.html

OWASP. (2010). *Category:OWASP Top Ten Project*. Retrieved February 7, 2013, from OWASP: The Open Web Application Security Project: https://www.owasp.org/index.php/Top_10

OWASP. (2012, November 25). Retrieved February 1, 2013, from OWASP: https://www.owasp.org/index.php/Main_Page

OWASP. (2013, January 25). *XSS Filter Evasion Cheat Sheet.* Retrieved February 1, 2013, from OWASP: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Shar, & Tan. (2012). Defending against cross-site scripting attacks. *Computer*, 55-62.

Su, Z., & Wassermann, G. (2006). *The Essence of Command Injection Attacks in Web Applications.* University of California, Davis.

The Flex Project. (2008). *flex: The Fast Lexical Analyzer*. Retrieved 2012, from SourceForge: http://flex.sourceforge.net/

Wassermann, G., & Su, Z. (2007). *Sound and Precise Analysis of Web Applicatiosn for Injection Vulnerabilities.* University of California, Davis.

Yang, E. (2012). Retrieved February 1, 2013, from HTML Purifier: http://htmlpurifier.org/

Appendix A : **Scanner Configuration File**

The MSL script that generates the token definitions for Flex copies this file verbatim into the output file before printing the procedurally generated content.

```
%option noyywrap
%{
#include <ctype.h>
#include "sfh.tab.h"
#include <stdbool.h>
#define DEBUG 0

bool _inOpenTag = false;
bool _inQuote = false;
int _lastID = ERROR;
int parseID();
void printDebug();
void upperString(char*);
int main();
%}
SYMBOL          ("!"|"#"|"$"|"%"|"&"|"'"|"("|")"|"*"|"+"|","|"-
"|"."|"/"|":"|";"|"="|"?"|"@"|"["|"\\"|"]"|"^"|"_"|"`"|"{"|"|"|"}")
NUM                     [0-9]
ALPHA           [a-zA-z]
ALNUM           {ALPHA}|{NUM}
SPACE           " "|"\t"|"\r"|"\n"
ID                      {ALPHA}{ALNUM}*
PCDATA          ({ALNUM}|{SYMBOL})({SPACE}*{ALNUM}|{SPACE}*{SYMBOL})*
URI                     ("http://"|"https://"|"/"|"./")({ALNUM}|{SYMBOL})*
DOCTYPE         "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\"
\"http://www.w3.org/TR/html4/strict.dtd\">"
%%
{DOCTYPE}       {
                                printDebug("DOCTYPE");
                                return DOCTYPE;
                        }
"<"                     {
                                printDebug("LT");
                                _inOpenTag = true;
                                return '<';
                        }
">"                     {
                                printDebug("GT");
                                _inOpenTag = false;
                                return '>';
                        }
{URI}           {
                                if (_lastID == SRCID) {
                                        _lastID = ERROR;
                                        printDebug("URI");
                                        return URI;
                                }
                                else {
```

```
                                        REJECT;
                                }
                        }
{PCDATA}        {
                                if (_inQuote || !_inOpenTag) { printDebug("PCDATA");
return PCDATA; }
                                else {
                                        //printf("Reject CDATA\n");
                                        REJECT;
                                }
                        }
{ID}            {
                                if (_inQuote) {
                                        //printf("Reject ID\n");
                                        REJECT;
                                } else {
                                        printDebug("ID");
                                        _lastID = parseID();
                                        return _lastID;
                                }
                        }
"\""            {
                                _inQuote = ! _inQuote;
                                printDebug("\"");
                                return 34;
                        }

"="                     {
                                printDebug("=");
                                if (_inOpenTag) { return '='; }
                                else { REJECT; }
                        }
"/"                     {
                                printDebug("/");
                                if (_inOpenTag) { return '/'; }
                                else { REJECT; }
                        }
{SPACE}         ;
.                       {
                                printDebug("ERROR");
                                return ERROR;
                        }
%%
void printDebug(char* token) {
        if (DEBUG) {
                printf("%s: %s\n",token,yytext);
        }
}
void upperString(char* in) {
        int pos = 0;
        for (pos = 0; in[pos] != '\0'; pos++) {
                in[pos] = toupper(in[pos]);
        }
}
```

Appendix B : **Grammar Configuration Files**

The MSL script that generates the grammar definition effectively procedurally generates token definitions, copies the first file verbatim, procedurally generates rewrite rules, and then copies the second file verbatim.

```
%defines
%error-verbose
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
%}
%token PCDATA;
%token URI;
%token ERROR;
%token DOCTYPE;
%%
START : DOCTYPE HTML
```

```
%%
void yyerror(const char *s) {
        printf("%s\n",s);
        exit(-1);
}

int main (void) {
        return yyparse ();
}
```

Appendix C : **Token Definitions**

The following is the input to Flex, which generates a token scanner. The `parseID` method is abbreviated to save space, given that it repeats in a predictable manner.

```
%option noyywrap
%{
#include <ctype.h>
#include "sfh.tab.h"
#include <stdbool.h>
#define DEBUG 0

bool _inOpenTag = false;
bool _inQuote = false;
int _lastID = ERROR;
int parseID();
void printDebug();
void upperString(char*);
int main();
%}
SYMBOL          ("!"|"#"|"$"|"%"|"&"|"'"|"("|")"|"*"|"+"|","|"-
"|"."|"/"|":"|";"|"="|"?"|"@"|"["|"\\"|"]"|"^"|"_"|"`"|"{"|"|"|"}")
NUM                     [0-9]
ALPHA           [a-zA-z]
ALNUM           {ALPHA}|{NUM}
SPACE           " "|"\t"|"\r"|"\n"
ID                      {ALPHA}{ALNUM}*
PCDATA          ({ALNUM}|{SYMBOL})({SPACE}*{ALNUM}|{SPACE}*{SYMBOL})*
URI                     ("http://"|"https://"|"/"|"./")({ALNUM}|{SYMBOL})*
DOCTYPE         "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">"
%%
{DOCTYPE}       {
                                printDebug("DOCTYPE");
                                return DOCTYPE;
                        }
"<"                     {
                                printDebug("LT");
                                _inOpenTag = true;
                                return '<';
                        }
">"                     {
                                printDebug("GT");
                                _inOpenTag = false;
                                return '>';
                        }
{URI}           {
                                if (_lastID == SRCID) {
                                        _lastID = ERROR;
                                        printDebug("URI");
```

```
                                        return URI;
                                }
                                else {
                                        REJECT;
                                }
                        }
{PCDATA}        {
                                if (_inQuote || !_inOpenTag) { printDebug("PCDATA");
return PCDATA; }
                                else {
                                        //printf("Reject CDATA\n");
                                        REJECT;
                                }
                        }
{ID}            {
                                if (_inQuote) {
                                        //printf("Reject ID\n");
                                        REJECT;
                                } else {
                                        printDebug("ID");
                                        _lastID = parseID();
                                        return _lastID;
                                }
                        }
"\""            {
                                _inQuote = ! _inQuote;
                                printDebug("\"");
                                return 34;
                        }

"="                     {
                                printDebug("=");
                                if (_inOpenTag) { return '='; }
                                else { REJECT; }
                        }
"/"                     {
                                printDebug("/");
                                if (_inOpenTag) { return '/'; }
                                else { REJECT; }
                        }
{SPACE}         ;
.                       {
                                printDebug("ERROR");
                                return ERROR;
                        }
%%
void printDebug(char* token) {
        if (DEBUG) {
                printf("%s: %s\n",token,yytext);
        }
}
void upperString(char* in) {
        int pos = 0;
        for (pos = 0; in[pos] != '\0'; pos++) {
                in[pos] = toupper(in[pos]);
        }
}
int parseID() {
        upperString(yytext);
        if (strcmp(yytext,"TT") == 0) return TTID; else
        if (strcmp(yytext,"I") == 0) return IID; else
(…)
        if (strcmp(yytext,"DIR") == 0) return DIRID; else
```

```
        return ERROR;
}
```

Appendix D : **Grammar Definition**

The following is the input to Bison, which produces the parser. The token definitions and reproduction rules are abbreviated to save space, given that they repeat in a predictable pattern.

```
%token DIRID
%token LANGID
(…)
%token TTID
%defines
%error-verbose
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
%}
%token PCDATA;
%token URI;
%token ERROR;
%token DOCTYPE
%%
START : DOCTYPE HTML
HTMLATTRIBUTES : HTMLATTRIBUTES DIRID '=' '\"' PCDATA '"' | HTMLATTRIBUTES LANGID
'=' '\"' PCDATA '"' | ;
NOSCRIPTATTRIBUTES : NOSCRIPTATTRIBUTES DIRID '=' '\"' PCDATA '"' |
NOSCRIPTATTRIBUTES LANGID '=' '\"' PCDATA '"' | NOSCRIPTATTRIBUTES TITLEID '='
'\"' PCDATA '"' | NOSCRIPTATTRIBUTES CLASSID '=' '\"' PCDATA '"' |
NOSCRIPTATTRIBUTES IDID '=' '\"' PCDATA '"' | ;
(…)
TTATTRIBUTES : TTATTRIBUTES DIRID '=' '\"' PCDATA '"' | TTATTRIBUTES LANGID '='
'\"' PCDATA '"' | TTATTRIBUTES TITLEID '=' '\"' PCDATA '"' | TTATTRIBUTES CLASSID
'=' '\"' PCDATA '"' | TTATTRIBUTES IDID '=' '\"' PCDATA '"' | ;
HTML : '<' HTMLID HTMLATTRIBUTES '>' HTMLSUBELEMENTS '<' '/' HTMLID '>' ;
HTMLSUBELEMENTS : HTMLSUBELEMENTS HEAD | HTMLSUBELEMENTS BODY | ;
NOSCRIPT : '<' NOSCRIPTID NOSCRIPTATTRIBUTES '>' NOSCRIPTSUBELEMENTS '<' '/'
NOSCRIPTID '>' ;
NOSCRIPTSUBELEMENTS : NOSCRIPTSUBELEMENTS P | NOSCRIPTSUBELEMENTS H1 |
NOSCRIPTSUBELEMENTS H2 | NOSCRIPTSUBELEMENTS H3 | NOSCRIPTSUBELEMENTS H4 |
NOSCRIPTSUBELEMENTS H5 | NOSCRIPTSUBELEMENTS H6 | NOSCRIPTSUBELEMENTS UL |
NOSCRIPTSUBELEMENTS OL | NOSCRIPTSUBELEMENTS PRE | NOSCRIPTSUBELEMENTS DL |
NOSCRIPTSUBELEMENTS DIV | NOSCRIPTSUBELEMENTS NOSCRIPT | NOSCRIPTSUBELEMENTS
BLOCKQUOTE | NOSCRIPTSUBELEMENTS FORM | NOSCRIPTSUBELEMENTS HR |
NOSCRIPTSUBELEMENTS TABLE | NOSCRIPTSUBELEMENTS FIELDSET | NOSCRIPTSUBELEMENTS
ADDRESS | ;
(…)
TTSUBELEMENTS : TTSUBELEMENTS PCDATA | TTSUBELEMENTS TT | TTSUBELEMENTS I |
TTSUBELEMENTS B | TTSUBELEMENTS BIG | TTSUBELEMENTS SMALL | TTSUBELEMENTS EM |
TTSUBELEMENTS STRONG | TTSUBELEMENTS DFN | TTSUBELEMENTS CODE | TTSUBELEMENTS SAMP
| TTSUBELEMENTS KBD | TTSUBELEMENTS VAR | TTSUBELEMENTS CITE | TTSUBELEMENTS ABBR
```

```
| TTSUBELEMENTS ACRONYM | TTSUBELEMENTS A | TTSUBELEMENTS IMG | TTSUBELEMENTS BR |
TTSUBELEMENTS MAP | TTSUBELEMENTS Q | TTSUBELEMENTS SUB | TTSUBELEMENTS SUP |
TTSUBELEMENTS SPAN | TTSUBELEMENTS BDO | TTSUBELEMENTS INPUT | TTSUBELEMENTS
SELECT | TTSUBELEMENTS TEXTAREA | TTSUBELEMENTS LABEL | TTSUBELEMENTS BUTTON | ;
%%
void yyerror(const char *s) {
        printf("%s\n",s);
        exit(-1);
}

int main (void) {
        return yyparse ();
}
```

Appendix E : **MSL Scripts**

Below are the entirety of the MSL scripts used in SFH4.

```
;-----STEP 1-----
removeComments {
  var %line $1

  var %result

  while (%line != $null) {
    var %pos1 $pos(%line,--,1)
    var %pos2 $pos(%line,--,2)

    if (%pos1 == $null) {
      var %result %result $+ %line
      var %line $null
    }
    elseif (%pos2 == $null) {
      var %result %result $+ $left(%line,$calc(%pos1 - 1))
      var %line $null
    }
    else {
      var %result %result $+ $left(%line,$calc(%pos1 - 1))
      var %line $right(%line,$calc(%pos2 * -1 - 1))
    }
  }

  return $replace(%result,$chr(9),$chr(32))
}

xmlToCsv {
  tokenize 32 $1-

  if ($1 == <!ATTLIST) {
    var %buffer $2 , $+ $left($3-,-1)
  }
  elseif ($1 == <!ELEMENT) {
    var %buffer $2 , $+ $left($5-,-1)
  }
  elseif ($1 == <!ENTITY) {
    var %buffer $3 , $+ $left($4-,-1)
  }
  return %buffer
}

parseItems {
  var %inFile $1
  var %entityOut $2
  var %attributeOut $3
  var %elementOut $4

  var %removeNewLines $true
```

```
  write -c %entityOut
  write -c %attributeOut
  write -c %elementOut

  var %inItem 0
  var %buffer
  var %outFile $null

  var %curLine 1
  var %totalLines $lines(%inFile)
  while (%curLine <= %totalLines) {
    var %line $read(%inFile,n,%curLine)

    if (<!* iswm %line) {
      %inItem = 1
      if (ENTITY isin $gettok(%line,1,32)) {
        var %outFile %entityOut
      }
      elseif (ATTLIST isin $gettok(%line,1,32)) {
        var %outFile %attributeOut
      }
      elseif (ELEMENT isin $gettok(%line,1,32)) {
        var %outFile %elementOut
      }
      else {
        var %outFile $null
      }
    }
    if (%inItem == 1) {
      %buffer = %buffer %line
    }
    if (*> iswm %line) {
      if (%outFile) {
        var %buffer $removeComments(%buffer)
        var %buffer $xmlToCsv(%buffer)
        write %outFile %buffer
        var %outFile $null
      }
      %inItem = 0
      %buffer = $null
    }
    %curLine = %curLine + 1
  }
}

;-----STEP 2-----
csvContainsVariable {
  var %line $1

  if (*%*;* iswm %line) {
    return $true
  }
  return %false
}

findVariable {
  var %line $1
  var %num $2

  if (%num > 0) {
    var %pos1 $pos(%line,%,%num)
    var %substr $right(%line,$calc(-1 * %pos1))
```

```
    var %pos2 $pos(%substr,$chr(59)%num)
    if (%pos2 == $null) {
      return $false
    }
    return $mid(%line,$calc(%pos1 + 1),$calc(%pos2 - 1))
  }
}

evaluateEntities {
  var %inFile $1
  var %tempFile $2
  var %outFile $3

  write -c %outFile
  write -c %tempFile

  var %iterations 1
  var %replacements 1
  var %writingToOutFile $true
  while (%replacements > 0) {
    var %replacements 0

    var %lines $lines(%inFile)
    var %s 1

    while (%s <= %lines) {
      var %line $read(%inFile,n,%s)
      var %name hgd.dtd

      if (*%*;* iswm %line) {
        var %replacements 1

        var %variable $findVariable(%line,1)
        var %formattedVariable $+(%,%variable,$chr(59))
        var %rawLookup $read(%inFile,ns,%variable)
        var %lookup $left($right(%rawLookup,-2),-1)
        var %replacement $replace(%line,%formattedVariable,%lookup)

        write %tempFile %replacement
      }
      else {
        write %tempFile %line
      }
      inc %s
    }
    .remove %outFile
    .rename %tempFile %outFile
    var %inFile %outFile

    inc %iterations
    if (%iterations == 15) {
      echo -s Halted; non-terminating loop
      halt
    }
  }
}

;-----STEP 3-----
evaluateElements {
  var %inFile $1
  var %refFile $2
  var %outFile $3
```

```
  write -c %outFile

  var %s 1
  var %lines $lines(%inFile)
  while (%s <= %lines) {
    var %line $read(%inFile,n,%s)

    while (*%*;* iswm %line) {
      var %variable $findVariable(%line,1)
      var %formattedVariable $+(%,%variable,$chr(59))
      var %rawLookup $read(%refFile,ns,%variable)
      var %lookup $left($right(%rawLookup,-2),-1)
      var %line $replace(%line,%formattedVariable,%lookup)
    }
    var %line $replace(%line,$chr(32),!)
    tokenize 44 %line

    var %start $remove($1,$chr(40),$chr(41),!)
    var %value $replacex($2-,$chr(32),$chr(124),!,$chr(32))
    var %left $null
    var %right $null

    var %cp $pos(%value,$chr(41),0)
    if (%cp == 2) {
      if ($pos(%value,$chr(41),1) < $pos(%value,$chr(40),2)) {
        var %cp $pos(%value,$chr(41),1)
      }
      else {
        var %cp $pos(%value,$chr(41),2)
      }
      inc %cp
    }
    elseif (%cp == 1) {
      var %cp $calc( $pos(%value,$chr(41),1) + 1)
    }
    else {
      var %cp $len(%value)
    }

    var %left $remove($left(%value,%cp),$chr(40),$chr(41),*,+,?,$chr(35),$chr(32))
    var %left $replace(%left,&,|)
    var %rightc $calc($len(%value) - %cp)
    if (%rightc > 0) {
      var %right
$remove($right(%value,%rightc),$chr(40),$chr(41),*,?,$chr(35),$chr(32))
      var %symbol $left(%right,1)
      var %right $right(%right,-1)
      var %p 1
      var %tot $numtok(%right,124)
      while (%p <= %tot) {
        var %tok $gettok(%right,%p,124)
        if (%symbol == +) {
          var %left $+(%left,|,%tok)
        }
        elseif (%symbol == -) {
          var %left $remtok(%left,%tok,1,124)
        }
        inc %p
      }
    }
    var %p 1
    var %tot $numtok(%start,124)
    while (%p <= %tot) {
```

```
      var %line $gettok(%start,%p,124) $+ , $+ %left
      write %outFile %line
      inc %p
    }
    inc %s
  }
}

;-----STEP 4-----
simplifyAttributes {
  var %right $1-

  var %op $chr(40)
  var %cp $chr(41)

  var %buffer
  var %p 1
  var %tot $numtok(%right,32)
  var %ttype 1
  var %name
  var %valstart
  var %val
  var %wait $false
  while (%p <= %tot) {
    var %tok $gettok(%right,%p,32)
    if (%ttype == 1) {
      var %name %tok
      inc %ttype
      var %valstart $calc(%p + 1)
    }
    elseif (%ttype == 2) {
      if ($+(%op,*) iswm %tok) {
        var %wait $true
      }
      if ($+(*,%cp) iswm %tok) {
        var %wait $false
      }
      if (!%wait) {
        var %val $gettok(%right,$+(%valstart,-,%p),32)
        ;Overwriting %val for simplicity of Flex/Bison parsing.  Prefix val with
"Keep:" to keep it
        if (keep:* iswm %val) {
          var %val $gettok(%val,2,58)
        }
        else {
          var %val PCDATA
        }
        var %buffer %buffer $+ %name %val $+ $chr(44)
        inc %ttype
      }
    }
    elseif (%ttype == 3) {
      var %ttype 1
    }
    else {
      echo -s Error: Unknown token index
      halt
    }

    inc %p
  }
  return %buffer
}
```

```
evaluateAttributes {
  var %inFile $1
  var %refFile $2
  var %outFile $3

  var %comma $chr(44)
  var %op $chr(40)
  var %cp $chr(41)

  write -c %outFile

  var %s 1
  var %lines $lines(%inFile)
  while (%s <= %lines) {
    var %line $read(%inFile,n,%s)

    ;Evaluate entities
    while (*%*;* iswm %line) {
      var %variable $findVariable(%line,1)
      var %formattedVariable $+(%,%variable,$chr(59))
      var %rawLookup $read(%refFile,ns,%variable)
      var %lookup $left($right(%rawLookup,-2),-1)
      var %line $replace(%line,%formattedVariable,%lookup)
    }
    var %left $remove($gettok(%line,1,44),$chr(32),$chr(40),$chr(41))
    var %right $simplifyAttributes($gettok(%line,2-,44))

    var %tot $numtok(%left,124)
    var %p 1
    while (%p <= %tot) {
      var %tok $gettok(%left,%p,124)
      var %line $left($+(%tok,$chr(44),%right),-1)
      var %line $replace(%line,$+($chr(32),%comma,$chr(32)),%comma)

      write %outFile %line
      inc %p
    }

    inc %s
  }
}

;-----STEP 5-----
filterElements {
  var %line $1
  var %elementsList $2

  var %element $gettok(%line,1,44)
  if ($istok(%elementsList,%element,44)) {
    return $null
  }
  else {
    var %subs $gettok(%line,2,44)

    var %n $numtok(%elementslist,44)
    while (%n > 0) {
      var %rmtok $gettok(%elementsList,%n,44)
      var %subs $remtok(%subs,%rmtok,0,124)

      dec %n
    }
    return %element $+ , $+ %subs
```

```
  }
}

filterAttributes {
  var %line $1
  var %attributesList $2
  var %elementsList $3

  var %element $gettok(%line,1,44)
  if ($istok(%elementsList,%element,44)) {
    return $null
  }
  else {
    var %attributes $gettok(%line,2-,44)

    var %n $numtok(%attributesList,44)
    while (%n > 0) {
      var %rmtok $gettok(%attributesList,%n,44)
      var %attributes $remtok(%attributes,%rmtok,0,44)
      dec %n
    }
    return %element $+ , $+ %attributes
  }
}

redefineElement {
  var %attributesFile $1
  var %element $2
  var %attribute $3
  var %oldValue $4
  var %newValue $5

  var %line $read(%attributesFile,nw,$+(%element,$chr(44),*))
  var %n $readn
  if (%n < 1) { halt }

  var %atts $gettok(%line,2-,44)
  var %atts $remtok(%atts,%attribute %oldValue,0,44)
  var %atts $addtok(%atts,%attribute %newValue,44)

  var %line %element $+ $chr(44) $+ %atts

  write -l $+ %n %attributesFile %line
}

removeItems {
  ;INPUT VARIABLES
  var %elementsFile $1
  var %attributesFile $2
  var %elementsList $3
  var %attributesList $4

  var %bar $chr(124)

  ;ELEMENTS FILE
  var %p $lines(%elementsFile)
  while (%p > 0) {
    var %line $read(%elementsFile,n,%p)
    var %line $filterElements(%line,%elementsList)
    if (%line == $null) {
      write -dl $+ %p %elementsFile
    }
    else {
```

```
      write -l $+ %p %elementsFile %line
    }
    dec %p
  }

  ;ATTRIBUTES FILE
  var %p $lines(%attributesFile)
  while (%p > 0) {
    var %line $read(%attributesFile,n,%p)
    var %line $filterAttributes(%line,%attributesList,%elementsList)
    if (%line == $null) {
      write -dl $+ %p %attributesFile
    }
    else {
      write -l $+ %p %attributesFile %line
    }
    dec %p
  }
}


;-----STEP 6-----
parseAttributesFile {
  ;GOALS
  ;1) enumerate attribute IDs
  ;2) write attribute rewrites in Bison

  ;INPUT VARIABLES
  var %line $1
  var %idList $2
  var %writeCmd $3

  ;STRING MANIPULATION VARIABLES
  var %bar $chr(124)
  var %element $gettok(%line,1,44)
  var %elementAttributes $+(%element,ATTRIBUTES) :

  ;ITERATE OVER ATTRIBUTE DECLARATIONS
  var %s $numtok(%line,44)
  while (%s > 1) {
    ;STRING MANIPULATION VARIABLES
    var %tok $gettok(%line,%s,44)
    tokenize 32 %tok
    var %upper $remove($upper($1),-)
    var %val $2
    var %id %upper $+ ID

    ;Achieves Goal 1: ENUMERATE IDS
    var %idList $addtok(%idList,%upper,44)

    ;Achieves Goal 2: ATTRIBUTE DECLARATIONS
    var %elementAttributes %elementAttributes $+(%element,ATTRIBUTES) %id '=' '\"'
%val '"' %bar

    dec %s
  }
  ;Achieves Goal 2: ATTRIBUTE DECLARATIONS
  %writeCmd %elementAttributes ;

  return %idList
}

parseElementsFile {
```

```
  ;GOALS:
  ;1) enumerate element IDs
  ;2) write subelements rewrites in Bison
  ;3) write element declaration rewrites in Bison

  ;INPUT VARIABLES
  var %line $1
  var %idList $2
  var %writeCmd $3

  ;STRING MANIPULATION VARIABLES
  var %bar $chr(124)
  var %element $remove($gettok(%line,1,44),$chr(32))
  var %upper $upper(%element)
  var %id $remove($+(%upper,ID),-)
  var %sub $remove($gettok(%line,2,44),$chr(32))
  var %sub $replace(%sub,PCDATA,CDATA,CDATA,PCDATA)
  var %text-sub $+(%upper,SUBELEMENTS)
  var %text-att $+(%upper,ATTRIBUTES)

  ;SETUP FOR CLOSE TAGS
  var %close $null
  if (%sub != EMPTY) {
    var %close '<' '/' %id '>'
  }
  else {
    var %sub $null
  }

  ;Achieves Goal 3: ELEMENT DECLARATION
  %writeCmd %upper : '<' %id %text-att '>' $iif(%sub,%text-sub %close,) ;

  ;Achieves Goal 2: SUBELEMENT DECLARATION
  if (%sub) {
    %writeCmd %text-sub : %text-sub $replace(%sub,%bar,$+($chr(32),%bar %text-
sub,$chr(32))) %bar ;
  }

  ;Achieves Goal 1: ID DECLARATION
  return $addtok(%idlist,%upper,44)
}

makeInputFiles {
  ;INPUT VARIABLES
  var %elementsFile $1
  var %attributesFile $2
  var %bisonPrefix $3
  var %bisonSuffix $4
  var %flexPrefix $5
  var %bisonOut $6
  var %flexOut $7
  var %tempFile $8

  ;GOALS:
  ;1) Attribute ID tokens in Bison
  ;2) Element ID tokens in Bison
  ;3) Element declaration rewrites in Bison
  ;4) Attribute rewrites in Bison
  ;5) Subelements rewrites in Bison
  ;6) parseID() function in Flex

  ;STRING MANIPULATION VARIABLES
  var %nl $chr(10)
```

```
  var %per $chr(37)
  var %bar $chr(124)
  var %tab $chr(9)

  ;CLEAR OUTPUT FILES
  write -c %bisonOut
  write -c %flexOut
  write -c %tempFile

  ;SET STARTING CONTENT FOR OUTPUT FILES
  if ($isFile(%bisonPrefix)) {
    .copy -o %bisonPrefix %bisonOut
  }
  if ($isFile(%flexPrefix)) {
    .copy -o %flexPrefix %flexOut
  }

  ;PARSE ATTRIBUTES FILE
  var %idList
  var %t $lines(%attributesFile)
  while (%t > 0) {
    var %line $read(%attributesFile,n,%t)

    ;Achives Goals 1 AND 4
    var %idList $parseAttributesFile(%line,%idList,write %tempFile)

    dec %t
  }

  ;PARSE SUBELEMENTS FILE
  var %t $lines(%elementsFile)
  while (%t > 0) {
    var %line $read(%elementsFile,n,%t)

    ;Achieves Goals 2, 3, AND 5
    var %idList $parseElementsFile(%line,%idList,write %tempFile)

    dec %t
  }

  ;Achieves goals 1, 2, AND 6
  write %flexOut int parseID() $chr(123)
  write %flexOut %tab $+ upperString(yytext);

  ;Parse through collected attributes
  var %tot $numtok(%idList,44)
  while (%tot > 0) {
    var %upper $gettok(%idList,%tot,44)
    var %id %upper $+ ID

    ;Bison section: Attribute tokens
    write -il1 %bisonOut $+(%per,token) %id

    ;Flex section: parseAttribute() body
    write %flexOut %tab $+ if (strcmp(yytext," $+ %upper $+ ") == 0) return %id $+
; else

    dec %tot
  }
  write %flexOut %tab $+ return ERROR;
  write %flexOut $chr(125) %nl %nl

  .copy -a %tempFile %bisonOut
```

```
  .copy -a %bisonSuffix %bisonOut
}

;-----SUMMARY-----
generateSFH {
  ;Note: If %directory doesn't exist, mIRC will return errors
  var %directory $mircdir $+ sfh/
  var %dtd $+(%directory,strict.dtd)
  var %tempFile $+(%directory,temp.txt)

  var %entitiesCSV $+(%directory,sfh.entities.csv)
  var %elementsCSV $+(%directory,sfh.elements.csv)
  var %attributesCSV $+(%directory,sfh.attributes.csv)

  var %evaluatedEntities $+(%directory,sfh.entitiesEvaluated.csv)
  var %evaluatedAttributes $+(%directory,sfh.attributesEvaluated.csv)
  var %evaluatedElements $+(%directory,sfh.elementsEvaluated.csv)

  var %elementsList $replace(base link meta object param script
style,$chr(32),$chr(44))
  var %attributesList onclick PCDATA,ondblclick PCDATA,onmousedown
PCDATA,onmouseup PCDATA,onmouseover PCDATA,onmousemove PCDATA,onmouseout
PCDATA,onkeypress PCDATA,onkeydown PCDATA,onkeyup PCDATA,action PCDATA,profile
PCDATA,onfocus PCDATA,onblur PCDATA,style PCDATA

  var %bisonPrefix $+(%directory,sfh.preBison.txt)
  var %bisonSuffix $+(%directory,sfh.postBison.txt)
  var %flexPrefix $+(%directory,sfh.preFlex.txt)
  var %bisonOut $+(%directory,sfh.y)
  var %flexOut $+(%directory,sfh.l)

  var %totalTime $ctime

  echo -s Starting Step 1
  var %time $ctime
  noop $parseItems(%dtd,%entitiesCSV,%attributesCSV,%elementsCSV)
  var %duration $calc($ctime - %time)
  echo -s Done! $duration(%duration)

  echo -s Starting Step 2
  var %time $ctime
  noop $evaluateEntities(%entitiesCSV,%tempFile,%evaluatedEntities)
  var %duration $calc($ctime - %time)
  echo -s Done! $duration(%duration)

  echo -s Starting Step 3
  var %time $ctime
  noop $evaluateElements(%elementsCSV,%evaluatedEntities,%evaluatedElements)
  var %duration $calc($ctime - %time)
  echo -s Done! $duration(%duration)

  echo -s Starting Step 4
  var %time $ctime
  noop $evaluateAttributes(%attributesCSV,%evaluatedEntities,%evaluatedAttributes)
  var %duration $calc($ctime - %time)
  echo -s Done! $duration(%duration)

  echo -s Starting Step 5
  var %time $ctime
  noop
$removeItems(%evaluatedElements,%evaluatedAttributes,%elementsList,%attributesList
)
  noop $redefineElement(%evaluatedAttributes,IMG,src,PCDATA,URI)
```

```
  var %duration $calc($ctime - %time)
  echo -s Done! $duration(%duration)

  echo -s Starting Step 6
  var %time $ctime
  noop
$makeInputFiles(%evaluatedElements,%evaluatedAttributes,%bisonPrefix,%bisonSuffix,
%flexPrefix,%bisonOut,%flexOut,%tempFile)
  var %duration $calc($ctime - %time)
  echo -s Done! $duration(%duration)

  var %duration $calc($ctime - %totalTime)
  echo -s Total process: $duration(%duration)
}

isLetter {
  var %asc $asc($1)
  if ((( %asc >= $asc(A) ) && ( %asc <= $asc(Z) ) ) || (( %asc >= $asc(a) ) && (
%asc <= $asc(z) ))) {
    return $true
  }
  else return $false
}
```

Appendix F : **Sample PHP Class**

This is a sample PHP class to interface with the SFH4 parser. The code includes comments explaining how methods should be used.

```
<?php
class SFHscanner {
    // Private member variables for the executable file and the buffer file for
    //   input sent to the scanner.
    private $program_file;
    private $buffer_file;

    // Input:
    //   $program_file: The location of the scanner executable
    //   $buffer_file: The location of a file to use as the scanner's input buffer
    function __construct($program_file, $buffer_file) {
        $this->program_file = $program_file;
        $this->buffer_file = $buffer_file;
    }

    // Summary: Runs $input through the SFH scanner with no prior processing
    // Input:
    //   $input: The input to scan
    // Returns: true if pass, false if reject
    //
    public function raw_scan($input) {
        $returned = "";
        $output = "";
        $operator = " ";

        file_put_contents($this->buffer_file, $input);
        $argument = $this->buffer_file;
        $operator = " < ";
        $cmd = $this->program_file.$operator.$argument;

        $result = exec($cmd, $output, $returned);
        if($returned == 0) {
            $returned = true;
        } else {
            $returned = false;
        }
        return $returned;
    }

    // Summary: Runs $input through SFH scanner after surrounding it with
    //   context as specified in $nest_array
    // Input:
    //   $input: The input to scan
    //   $nest_array: The array must be number-indexed, starting at 0 and
    //     increasing sequentially.  The elements must be tag names in
    //     descending order of nesting.  That is, the first item should be
    //     'HTML' followed by the next tag down, and so on.
```

```
    // Returns: true if test page passes, else false
    public function context_scan($raw_input, $nest_array) {
        $dtd = '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">';
        $prefix = '';
        $suffix = '';
        for ($i = count($nest_array); $i > 0; $i--) {
            $item = $nest_array[$i - 1];
            if ($item != NULL) {
                $prefix = '<'.$item.'>'.$prefix;
                $suffix .= '</'.$item.'>';
            }
        }
        $input = $dtd.$prefix.$raw_input.$suffix;
        return $this->raw_scan($input);
    }
}
?>
```